Representing Object Models as Theories

Proceedings of the 10th Knowledge-Based Software Engineering Conference
Boston Massachusetts, November 12-15, 1995

Scott DeLoach, Paul Bailor, and Thomas Hartrum

Department of Electrical & Computer Engineering
Air Force Institute of Technology
2950 P Street, Wright-Patterson AFB, OH 45433-7765

Representing Object Models as Theories

Scott DeLoach, Paul Bailor, and Thomas Hartrum
Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB, Ohio 45433

Abstract

Although techniques for using formal specifications have been progressing, methods for developing formal specifications themselves have improved little. To alleviate this problem, we propose a parallel refinement approach to specification acquisition where the designer uses an object-oriented specification representation while an underlying specification composition system manipulates an equivalent theory-based specification. This paper presents the key to such a system – a theory-based object model. Our theory-based object model formally defines object-oriented concepts in terms of algebraic theories and category theory operations. The theory-based object model provides the basis for the translation of the specification from a semi-formal, object-oriented representation to a formal theory-based specification suitable for input to semi-automated software synthesis systems.

1 Introduction

Use of algebraic theories to represent software engineering knowledge has gained momentum during the last decade. Some of the most promising work is using theory-based specifications to drive software synthesis systems. A notable example of such a synthesis system, the Kestrel Interactive Development System (KIDS) [1], has yielded some exciting results. KIDS has been used to derive dozens of algorithms including a transportation scheduling algorithm for over 15,000 movements that was 78 percent faster and had 75 percent fewer delays than the best previously known algorithms [2]. A follow-on effort to KIDS, Specware [3], supports a systematic approach to the composition of theory-based specifications followed by their stepwise refinement into code. The basic synthesis steps are to 1) develop a domain theory for the problem to be solved, 2) create a specification describing the problem in the language of its domain theory, 3) apply specification refinements to construct a program-based model of the problem specification, 4) apply program optimizations, and 5) compile the program [1].

While systems such as KIDS and Specware have been making progress in software synthesis research (steps 3 and 4 above), research in the acquisition of formal specifications (steps 1 and 2) has not been keeping pace. Formal specification of software remains an intricate, manually intensive activity. Problems associated with specification acquisition include a lack of expertise in mathematical and logical concepts among software developers, an inability to effectively communicate formal specifications with end users, and the tendency of formal notations to restrict solution creativity [4]. Fraser et. al., suggest an approach to overcoming these problems via *parallel refinement* of semi-formal and formal specifications. In a parallel refinement approach, designers develop specifications using both semi-formal and formal representations, successively refining both representations in parallel [4].

This paper presents a theory-based model of object-oriented concepts, our initial research into developing a parallel refinement methodology based on object-orientation and theories. Our approach is to develop specifications using a graphically-based, object-oriented paradigm while an underlying system automatically translates the object-oriented representation into a theory-based specification. In the system, a user might develop an object-oriented specification using a knowledge-based "assistant" to help refine or restructure informal parts of the specification to meet the stringent requirements of our transformation system.

We start by looking at related work in Section 2. Our view of object-oriented concepts is described in Section 3, and our theory-based object model and translation from object-oriented to theory-based models is discussed in Section 4. Future work is discussed in Section 5 followed by conclusions in Section 6.

2 Related Work

There have been a number efforts designed to incorporate object-oriented concepts into formal specification languages. MooZ [5] and Object-Z [6] extend Z by adding object-oriented structures while maintaining its model-based semantics. Z++ [7] and OOZE [8] also extend Z but provide semantics based on algebra and category theory. Although these Z extensions provide enhanced structuring techniques, they do not provide improved specification acquisition methods. FOOPS [9] is an algebraic, object-oriented specification language based on OBJ3 [10]. Both FOOPS and OBJ3 focus on prototyping, and provide little support for specification acquisition. Some research has been directed toward improving specification acquisition by translating object-oriented specifications into formal specifications [11]; however, these techniques are based on Z and lack a strong notion of refinement from specification to code.

3 The Informal Object Model

Because a standard set of definitions does not yet exist for many terms and concepts in object-orientation, we selected a specific object-oriented model before defining our theory-based object model. We chose the Rumbaugh Object Modeling Technique [12] for its breadth of coverage, availability of tools, and usefulness for domain analysis. Rumbaugh uses three distinct views to describe a system: (1) the object model describes structural relationships between system objects, (2) the dynamic model describes interactions between system objects, and (3) the functional model describes how the system transforms data. Rumbaugh focuses on four major themes: classes, objects with state, relationships between classes, and communication between objects.

An object class describes a group of objects with similar properties, behavior, relationships to other objects, and semantics. Whereas classes describe objects, an object is the actual entity whose state is manipulated by class operations. An object class defines an object's attributes and operations. Attributes are data held by an object while operations are actions that may be performed on an object. One of the basic concepts distinguishing object-orientation from other encapsulation techniques is that every object has a state, and the effects of an object's operations are dependent on its state.

There are three common relationships between classes: inheritance, association, and aggregation. Inheritance is the mechanism of obtaining attributes and operations using a generalization-specialization structure. In general, a "subclass" inherits all the attributes and operations of a parent class and may add additional attributes and operations of its own. In the generalization-specialization structure, each object of a subclass is also an object of the superclass. Multiple inheritance is the mechanism by which a class can inherit features from more than one superclass.

An association is a group of links with a common structure and semantics where a link is defined as a physical or conceptual connection between object instances. The relationship between associations and links is similar to the relationship between classes and objects. An association is a template that defines what classes of objects may be connected as well as association attributes. Association attributes are attributes that do not belong to any one of the objects involved in a link, but exist only when there is a link between objects.

The final relationship, aggregation, is an important concept in object-orientation. Aggregation is a relationship between two classes where one class represents the entire assembly and the other class is "part-of" the assembly. Without aggregate objects, a system formed by combining subsystems cannot be modeled.

To incorporate objects into non-trivial systems, they must be able to communicate with each other. In Rumbaugh's methodology, objects communicate via events. An event is an instantaneous one-way transmission of information from one object to another.

4 Objects as Theories

This section presents our theory-based object model that formally defines object-oriented concepts and is the basis for translating an object-oriented specification into a theory-based specification. Concepts of algebraic theories and category theory necessary to define the object model are presented in Section 4.1. The actual theory-based object model is defined in Section 4.2 followed by a discussion of the translation process in Section 4.3.

4.1 Theory Fundamentals

Theory-based algebraic specification is concerned with (1) modeling system behavior using algebras (a collection of values and operations on those values) and axioms that characterize algebra behavior, and (2) composition of larger specifications from smaller specifications. Composition of specifications is accomplished via specification building operations defined by category theory constructs [13]. A *theory* is the set of all assertions that can be logically proved from the axioms of a given specification. Thus, a specification defines a theory and is termed a *theory presentation*.

In algebraic specifications, the structure of a specification is defined in terms of *sorts*, abstract collections of values, and *operations* over those sorts. This structure is called a *signature*. A signature describes the structure of a solution; however, a signature does not specify semantics. To specify semantics, the definition of a signature is extended with axioms defining the intended semantics of signature operations. A signature with associated axioms is called a *specification*. An example of a specification is shown in Figure 1.

```
spec  ARRAY  is
sorts  E, I, A
operations
    assign   :  A, I, E → A
    apply    :  A, I → E
axioms ∀ (i,j ∈ I, a ∈ A, e ∈ E)
    (i = j) ⇒ apply(assign(a,i,e),j) = e;
    (i ≠ j) ⇒ apply(assign(a,i,e),j) = apply(a,j)
end
```

Figure 1: Array specification

A specification allows us to formally define the internal structure of object classes (attributes and operations); however, they do not provide the capability to reason about relationships between object classes. To create theory-based algebraic specifications that parallel object-oriented specifications, the ability to define and reason about relationships between theories, similar to those used in object-oriented approaches (inheritance, aggregation, etc.), must be available. Category theory is an abstract mathematical theory used to describe the external structure of various mathematical systems [14] and is used here to describe relationships between specifications.

```
spec  FINITE-MAP  is
sorts  M, D, R
operations
   empty     :  → M
   update    :  M, D, R → M
   apply     :  A, D → R
   def?      :  A, D → Boolean
axioms ∀ (d1,d2 ∈ D, m ∈ M, r ∈ R)
   (d1 = d2) ⇒ apply(update(m,d2,r),d1) = r;
   (d1 ≠ d2) ⇒ apply(assign(m,d2,r),d1) = apply(m,d1);
   def?(update(m,d2,r),d1) = (d1 = d2) ∨ def?(m,d1);
   def?(empty,d1) = false
end
```

Figure 2: Finite map specification

A category consists of a collection of *C-objects* and *C-arrows* between objects such that (1) there is a C-arrow from each object to itself, (2) C-arrows are composable, and (3) arrow composition is associative. The most common example is the category **Set** where "C-objects" are sets "C-arrows" are functions between sets. Specifications with the correct "C-arrows" (specification morphisms) form the category **Spec** which is of great interest in our research. A *specification morphism*, σ, is a pair of functions that map sorts (σ_S) and operations (σ_Ω) from one specification to compatible sorts and operations of a second specification such that the axioms of the first specification are theorems of the second specification. Intuitively, specification morphisms define how one specification is embedded in another. An example of a morphism from *array* to *finite-map* (Figure 2) is shown below.

$$\sigma_\Omega = \{\text{assign} \mapsto \text{update, apply} \mapsto \text{apply}\}$$
$$\sigma_S = \{\text{A} \mapsto \text{M, I} \mapsto \text{D, E} \mapsto \text{R}\}$$

Specification morphisms comprise the basic tool for defining and refining specifications. Our toolset can be extended to allow the creation of new specifications from a set of existing specifications. Often two specifications derived from a common ancestor specification need to be combined. The desired combination consists of the unique parts of two specifications and some "shared part" common to both specifications (the part defined in the shared ancestor specification). This combining operation is a *colimit.*

Conceptually, the colimit is the "shared union" of a set of specifications based on the morphisms between the specifications. These morphisms define equivalence classes of sorts and operations. For example, if a morphism, σ, from specification A to specification B maps sort α to sort β, then α and β are in the same equivalence class and thus become a single sort in the colimit specification of A, B, and σ. The colimit operation creates a new specification, the *colimit specification*, and a specification morphism from each specification to the colimit specification. An example showing the relationship between a colimit and multiple inheritance is provided in Section 4.2.

From these basic tools (morphisms and colimits), we can construct specifications in a number of ways [13]. We can (1) build a specification from a signature and a set of axioms, (2) form the union of a set of specifications via a colimit, (3) rename sorts or operations via a specification morphism, and (4) parameterize specifications. Many of these methods are useful in translating object-oriented specifications into theory-based specifications. Detailed semantics of object-oriented concepts using specifications and category theory constructs are presented next.

4.2 A Formal Object Model

Object Class. We start the description of our theory-based object model by defining the basic building block of object-orientation – an object class. An object class is defined to be a theory presentation representing seven components: a class sort, additional sorts referenced in the theory, attributes, states, state attributes, methods, and events. The class sort is a special sort in the theory while the attributes, states, state attributes, methods, and events are all operations in the theory.

Definition 4.1 Object Class - *A class, C, is a signature, $\Sigma = < S, \Omega >$ and a set of axioms, Φ, over Σ (i.e., a theory presentation, or specification) where*

$$\begin{array}{rcl} S & = & \textit{a set of sorts including the class sort} \\ \Omega & = & \textit{a set of operations over } S \textit{ representing} \\ & & \textit{attributes, states, state attributes,} \\ & & \textit{methods, and events} \\ \Phi & = & \textit{a set of axioms over } \Sigma \end{array}$$

An object class may have many instances, each of which represents a unique object in the class. Each value in the class sort is a reference to a particular object in the class. Objects themselves are not explicitly represented in a class definition or specification. They, and their attribute values, are maintained external to the class definition.

Attributes are functions that return the value of data held by an object. State attributes differ from normal attributes. State attributes return data about an object's state where each state is defined as a unique constant. Attributes return data about an object; they do not modify it in any way. Methods are defined as functions that may modify an object's normal attributes. Events and attributes define the class interface. There are two types of events: incoming and outgoing. Incoming events are functions that may modify an object's state attributes and cause a method to be invoked. Outgoing events are operations that map to incoming events in other classes and are discussed in more detail in the section Object Communication. Each class also has a *new* event and *create* method used to create valid objects of the class. Figure 3 shows an example of a theory-based representation of an object class. The operations `date` and `bal` are attributes, `acct-state` is a state attribute, `create-acct`, `credit` and `debit`

```
class ACCT is
import Amnt, Date
class-sort Acct
sorts Acct-State
operations
    attr-equal : Acct, Acct → Boolean
attributes
    date : Acct → Date
    bal : Acct → Amnt
state-attributes
    acct-state : Acct → Acct-State
methods
    create-acct : Date → Acct
    credit : Acct, Amnt → Acct
    debit : Acct, Amnt → Acct
states
    ok : → Acct-State
    overdrawn : → Acct-State
events
    new-acct : Date → Acct
    deposit : Acct, Amnt → Acct
    withdrawal : Acct, Amnt → Acct
axioms ∀ (d ∈ Date, a, a1 ∈ Acct, x ∈ Amnt)
  % state uniqueness axioms
    ok ≠ overdrawn;
  % operation definitions
    attr-equal(a, a1) ⇒ date(a) = date(a1) ∧ bal(a) = bal(a1);
    date(create-acct(d)) = d ;
    bal(create-acct(d)) = 0 ;
    acct-state(new-acct(d)) = ok ∧ attr-equal(new-acct(d), create-acct(d));
  % method definitions
    bal(credit(a,x)) = bal(a) + x ;
    bal(debit(a,x)) = bal(a) - x ;
  % deposit event definition
    acct-state(a) = ok ⇒ acct-state(deposit(a,x)) = ok ∧ attr-equal(deposit(a,x), credit(a,x));
    acct-state(a) = overdrawn ∧ bal(a) + x ≥ 0 ⇒ acct-state(deposit(a,x)) = ok
                     ∧ attr-equal(deposit(a,x), credit(a,x));
    acct-state(a) = overdrawn ∧ bal(a) + x < 0 ⇒ acct-state(deposit(a,x)) = overdrawn
                     ∧ attr-equal(deposit(a,x), credit(a,x));
  % withdrawal event definition
    acct-state(a) = ok ∧ bal(a) ≥ x ⇒ acct-state(withdrawal(a,x)) = ok
                     ∧ attr-equal(withdrawal(a,x), debit(a,x));
    acct-state(a) = ok ∧ bal(a) < x ⇒ acct-state(withdrawal(a,x)) = overdrawn
                     ∧ attr-equal(withdrawal(a,x), debit(a,x));
    acct-state(a) = overdrawn ⇒ acct-state(withdrawal(a,x)) = overdrawn
                     ∧ attr-equal(withdrawal(a,x), a)
end-class
```

Figure 3: Object class axioms

are methods, `ok` and `overdrawn` are possible states of `acct-state`, and `new-acct`, `deposit` and `withdrawal` are incoming events.

The axioms of a class are used to define the semantics of operations. In general, axioms are defined by describing the effect of methods on attributes or the effect of events on state attributes. In the `ACCT` class, the balance of the account after invoking the `credit` method is defined by the axiom `bal(credit(a, x)) = bal(a) + x`; however, the invocation of the `credit` method is controlled by the value of the `acct-state` attribute and reception of a `deposit` event as defined in Figure 3.

Inheritance. Class inheritance plays an important role in object-orientation; however, the correct use of inheritance is not uniformly agreed upon. Many languages provide "ad-hoc" inheritance that allows a subclass to redefine or even remove attributes or methods inherited from its superclass. However, most authors see the necessity to restrict the amount of modification freedom in a subclass. Our model implements a *generalization-specialization* inheritance relationship that requires that the subclass only *extend* the features of its superclass. Liskov defines this desired effect as the "substitution property" where a subclass object can be freely substituted for a superclass object in any environment designed for the superclass object [15]. The subsort operator $<$ defines a subset relationship among sorts such that for two sorts, A and B, $A < B \Rightarrow A \subseteq B$.

Definition 4.2 Inheritance - *A class, D, is said to inherit from a class, C, if there exists a specification morphism from C to D such that the class sort of D is a subsort of the class sort of C.*

This definition basically states that all sorts and operations (attributes, methods, and events) from class C are embedded in class D and that a new sort, the class sort of D, is defined as a subsort of the class sort of C. The specification morphism ensures that the "substitution property" holds. Figure 4 shows the specification for a savings account class that inherits directly from the `ACCT`. The `import` statement includes all the sorts, operations, and axioms declared in the `ACCT` class directly into `SAcct` while the class sort declaration `SAcct < Acct` states that `SAcct` is a subclass of `Acct`, and as such, all operations and axioms that apply to an `Acct` object apply to a `SAcct` object.

Multiple Inheritance. Multiple inheritance requires a slight modification to our notion of inheritance. The set of superclasses must first be combined and then used to "inherit from".

Definition 4.3 Multiple Inheritance - *A class D multiply inherits from a collection of classes, $(C_1..C_n)$ if there exists a specification morphism from the colimit of $(C_1 \,..\, C_n)$ to D such that the class sort of D is a subsort of each of the class sorts of $(C_1 \,..\, C_n)$.*

The colimit operation allows us to combine any number of classes, along shared parts, to create a single specification with all the sorts, operations, and

class SAcct is
import ACCT, Rate
class-sort SAcct $<$ Acct
operations
    attr-equal : SAcct, SAcct $\rightarrow$ Boolean
attributes
    rate : SAcct $\rightarrow$ Rate
    int-date : SAcct $\rightarrow$ Date
methods
    create-sacct : Date $\rightarrow$ SAcct
    set-rate : SAcct, Date, Rate $\rightarrow$ SAcct
    comp-int : SAcct, Date $\rightarrow$ SAcct
events
    new-sacct : Date $\rightarrow$ SAcct
    rate-change : SAcct, Date, Rate $\rightarrow$ SAcct
    compute-interest : SAcct, Date $\rightarrow$ SAcct
axioms $\forall$ (d $\in$ Date, r $\in$ Rate, a, a1 $\in$ SAcct)
    ... axioms omitted ...
end-class

Figure 4: Savings class

axioms of the original classes. We can then extend the colimit specification with the definition of the new class sort. Thus to create an account that combines the features of a savings account with those of a checking account (which inherits from the `ACCT` class similar to `SACCT`), we take the colimit of classes `ACCT`, `SACCT`, `CACCT`, and morphisms from `ACCT` to `SACCT` and `CACCT` as shown in Figure 5 (an arrow labeled with an "i" represents an import morphism and a "c" represents a morphism formed by the colimit operation). A simple extension of the colimit specification with the class sort definition, `Comb-Acct < SAcct, CAcct`, yields the desired combined class where `Comb-Acct` is a subclass of both `SAcct` and `CAcct`.

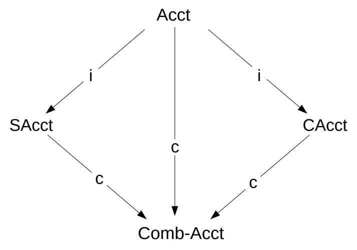

Figure 5: Colimit of accounts

Object-Valued Attributes. Because association, aggregation, and object communication require objects to be aware of other objects in the system, there must be some mechanism for objects to refer to each other. This mechanism is provided by *object-valued attributes* which are attributes that return references to other objects [9]. The sort of the value returned by an object-valued attribute is the class sort of the referenced object. Actually, an object-valued attribute returns a reference to an object, not the object itself. The use of references enables a system to have multiple object-valued attributes that reference the same object while maintaining only a single object.

Associations. In our model, we have chosen to model associations very generically, as sets of individual links.

Definition 4.4 Association *An association is defined as a tuple $\mathcal{A} =< \alpha, \lambda >$, where α is an object class whose class sort is a set of the class sort of λ, and λ is a class with two (or more) object-valued attributes.*

An association between the `ACCT` class and an unspecified `CUST` class is shown in Figure 6. The `CA-LINK` class plays the role of λ and has two object-valued attributes, `customer` and `account`, and a method to create new instances of the association. The `CUST-ACCT` class defines a set of `CA-Link` objects while the sorts `Accts` and `Custs` are sets of `Acct` and `Cust` objects. The axioms in `CUST-ACCT` define the multiplicity relationships between accounts and customers. In this case, there is exactly one customer per account while each customer may have one or more accounts. Associations with more than two classes are handled in a similar manner by simply adding additional object-valued attributes.

Aggregation. Although the object-valued attributes in `CA-Link` are named `cust` and `acct`, they have not been unified with the `CUST` and `ACCT` class sorts. Unification of these sorts requires a higher-level entity that describes how classes and associations interact. In our model, this higher-level entity is an aggregate class. An aggregate class has sub-objects (or components) and associations between them that implement the behavior of the aggregate. Once again, object-valued attributes are used to describe this relationship between objects.

Definition 4.5 Aggregate - *A class C is an* aggregate *of a collection of component classes, $(D_1..D_n)$, if there exists a specification morphism from the colimit of $(D_1..D_n)$ to C such that C has at least one corresponding object-valued attribute referencing each class in $(D_1..D_n)$.*

Therefore, an aggregate class allows us to combine a number of classes together via the colimit operation to specify system or sub-system level functionality. The colimit operation also gives us the capability to unify sorts and operations defined in separate classes and associations.

To create a system-level aggregate class, the colimit of all the object classes and associations within the system is taken. In the previous `CUST-ACCT` example, `CUST`, `ACCT`, and `CUST-ACCT` can be combined together into a system with the `Cust` sort from `CUST` and `CUST-ACCT`, and the `Acct` sort from `ACCT` and `CUST-ACCT`, being unified via morphisms that define their equivalence. In the colimit specification the `CUST-ACCT` association actually relates the `CUST` class to the `ACCT` class. New operations and axioms can be added to an extension of colimit specification that describe system-level interfaces and system behavior based on the operations and axioms in its components.

```
link CA-Link is
class-sort CA-Link
sorts Cust, Acct
operations
    attr-equal : CA-Link, CA-Link → Boolean
attributes
    customer : CA-Link → Cust
    account : CA-Link → Acct
methods
    create-ca-link : Cust, Acct → CA-Link
events
    new-ca-link : Cust, Acct → CA-Link
axioms ∀ (c ∈ Cust, a ∈ Acct, cl,cl1 ∈ CA-Link)
    attr-equal(cl, cl1) ⇒ customer(cl) = customer(cl1)
        ∧ account(cl) = account(cl1);
    customer(create-ca-link(c, a)) = c;
    account(create-ca-link(c, a)) = a;
    attr-equal(new-ca-link(c,a), create-ca-link(c,a))
end-link

association Cust-Acct is
link-class CA-Link
import Accts, Custs % sets of Acct and Cust
class-sort Cust-Acct
methods
    create-cust-acct : → Cust-Acct
    image : Cust-Acct, Cust → Accts
    image : Cust-Acct, Acct → Custs
events
    new-cust-acct : → Cust-Acct
axioms ∀ (ca, ca1 ∈ Cust-Acct, c ∈ Cust, a ∈ Acct)
    new-cust-acct() = create-cust-acct();
    create-cust-acct() = empty-set;
    size(image(ca, c)) ≥ 1;
    size(image(ca, a)) = 1;
    ... definition of image operations ...
end-association
```

Figure 6: Cust-Acct association

Object Communication. The model described so far is sufficient to describe classes and their relationships; however, it does not address communication between objects. In our model, each object is cognizant of the *events* that it generates. From an object's perspective, events are sent to some anonymous object. An event is then defined as an operation signature that maps to a method in some anonymous object class. The anonymous class sort and operation are defined in a separate *communication theory*. An example of a communication theory is shown in Figure 7. Notice that the communication theory is actually only a signature defining a class sort and an operation.

Each class that sends an event defines an object-valued attribute and operation signature for that event. A communication theory is then used to link the sending class to the receiving class via morphisms from the communication theory to the sending and receiving classes unifying the communication theory class sort and operation in the aggregate.

```
event EVENT-NAME is
class-sort Event-Sort
events
    event-name : Event-Sort → Event-Sort
end-class
```

Figure 7: Communication theory

4.3 Translation

The focus of our research to date has been to show that a graphically-based, object-oriented specification representation can be translated to and from a theory-based representation. Table 1 shows how Rumbaugh concepts map to theory-based concepts. There are some components of Rumbaugh's model specified informally via notes or data dictionaries that will require extending Rumbaugh's notion with additional formalisms. For example, if an object is required to send an event to a given class, there is no Rumbaugh notation for specifying which objects in the class should actually receive the event. The event may be sent to all objects in the class, objects with which certain links exist, or to objects specified by the system. While our model can capture such information, a direct translation from Rumbaugh's current notation is impossible due to its informality.

We have found it necessary to restrict Rumbaugh's notation in some cases. For instance, Rumbaugh allows operation semantics to be specified via tables, equations, pseudocode, natural language, or axiomatically. For obvious reasons, we require the operation semantics to be specified axiomatically. Also, in the dynamic model, Rumbaugh allows activities to be specified as occurring in a state or as actions occurring on the transitions (i.e., a combined Mealy-Moore machine); however, to simplify the translation process, we have restricted the dynamic model to a Mealy machine representation where all actions occur on transitions. This does not represent a semantic restriction since the equivalence of Mealy and Moore machines is well known [16].

Table 1: Rumbaugh to object model translation

Rumbaugh	**Theory-based Model**
classes	theory presentation
attributes	operation on class sort
operations	operation on class sort
constraints	axioms
object instances	logical variables
simple inheritance	morphism and subsort
multiple inheritance	colimit and subsort
aggregation	colimit and object-valued-attributes
multiplicity	axioms
associations	container of link objects
link	theory presentation
multiplicity	axioms
qualifier	attribute and axioms
link attributes	operations
link operations	operations
ordering	sequence of link objects
constraints	aggregate axioms
transition events	operations
parameters	operation parameters
actions	operations and axioms
output events	communication theories
state actions/activity	*see text*
processes	operations
operation definition	axioms
data flow	operations return values
control flow	communication theories
data store	object classes

5 Future Work

Figure 8 shows our concept of an object-oriented, theory-based parallel refinement specification acquisition system. The system assists in the development of theory-based domain models and system specifications. Although the designer is developing theory-based models, the designer interacts with the system via a conceptually simpler object-oriented representation. The critical component in the system is the Translator, that maps theory-based specifications to and from their object-oriented representations based on the theory-based object model discussed in this paper. The first step is to construct a domain model by specifying domain object classes and their associations graphically to define the structure of the domain. The object attributes and semantics are then specified algebraically or, preferably, graphically using behavior representing structures such as state charts and data flow diagrams that can be automatically translated into equivalent algebraic definitions. Once a domain model is developed, it is refined and used to create a system specification which is fed into a correctness preserving design (e.g., Specware) refinement mechanism that derives code satisfying the specification.

In this paper, we presented a theory-based object model that captures all the basic elements of an object-oriented specification. The next phase of our research is to precisely define the transformations necessary to automatically translate a system specified using Rumbaugh's object modeling notation into theories. This phase includes the definition of composition operations (based on category theory operations) that transform object-oriented specifications into corresponding theories. It also includes development of proof obligations necessary to show specification correctness and consistency, as well as completeness of the transformation process. An implementation of the transformation composition operations is planned to show the feasibility of theoretic results.

6 Conclusions

We have defined a theory-based object model that has the power and flexibility necessary to capture all

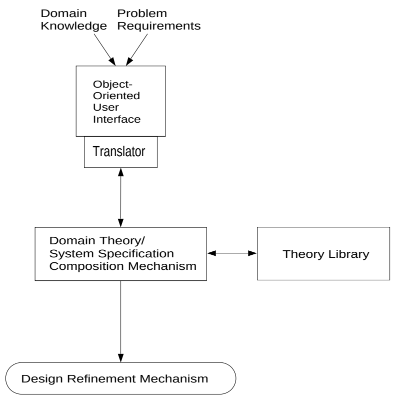
Domain Knowledge
Problem Requirements
Object-Oriented User Interface
Translator
Domain Theory/ System Specification Composition Mechanism
Theory Library
Design Refinement Mechanism

Figure 8: Parallel refinement structure

the fundamental elements of an object-oriented specification. Use of category theory operations in conjunction with algebraic specifications provides the capability to define and reason about the internal semantics of object classes as well as the structure and relationships of those classes. Our theory-based object model is the key to developing a parallel refinement approach where system specifications are developed using an object-oriented representation while an underlying specification composition system manipulates an equivalent theory-based specification. These theory-based system specifications may then be used as input to a semi-automated transformation system such as KIDS or Specware [1, 3] that produces code correctly implementing the specification.

Acknowledgments

This work has been supported by grants from Rome Laboratory, the National Security Agency, and the Air Force Office of Scientific Research.

References

[1] D. R. Smith, "KIDS - A Semi-automatic Program Development System," *IEEE Transactions of Software Engineering*, vol. 16, pp. 1024–1043, September 1990.

[2] D. R. Smith, "Transformational Approach to Transportation Scheduling," in *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pp. 60–68, IEEE, October 1993.

[3] Kestrel Institute, *Specware User Manual: Specware Version Core4*, October 1994.

[4] M. D. Fraser, K. Kumar, and V. K. Vaishnavi, "Strategies for Incorporating Formal Specifications," *Communications of the ACM*, vol. 37, pp. 74–86, October 1994.

[5] K. Lano and H. Houghton, "Specifying a Concept-recognition System in Z++," in *Object-Oriented Specification Case Studies* (K. Lano and H. Houghton, eds.), pp. 137–157, Prentice-Hall, 1994.

[6] D. Carrington *et al.*, "Object-Z: An Object-Oriented Extension to Z," in *Formal Description Techniques, II: Proceedings of the IFIP Second International Conference on Formal Description Techniques for Distributed Systems and Communications Protocol*, (Amsterdam), pp. 281–297, North-Holland, December 1989.

[7] K. Lano and H. Houghton, "A Comparative Description of Object-Oriented Specification Languages," in *Object-Oriented Specification Case Studies* (K. Lano and H. Houghton, eds.), pp. 20–54, Prentice-Hall, 1994.

[8] A. J. Alencar and J. A. Gougen, "Specification in OOZE with Examples," in *Object-Oriented Specification Case Studies* (K. Lano and H. Houghton, eds.), pp. 158–183, Prentice-Hall, 1994.

[9] J. A. Goguen and J. Meseguer, "Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics," in *Research Directions in Object-Oriented Programming* (B. Shriver and P. Wegner, eds.), pp. 417–477, MIT Press, 1987.

[10] J. A. Goguen and T. Winkler, "Introducing OBJ3," tech. rep., Computer Science Laboratory SRI International, 333 Ravenswood Ave, Menlo Park, CA, August 1988.

[11] T. C. Hartrum and P. D. Bailor, "Teaching formal extensions of informal-based object-oriented analysis methodologies," in *Software Engineering Education Proceedings*, (Pittsburgh, PA), Software Engineering Education, SEI, Software Engineering Institute (SEI), Jan. 1994.

[12] J. Rumbaugh *et al.*, *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1991.

[13] Y. V. Srinivas, "Algebraic Specification: Syntax, Semantics, Structure," tech. rep., Department of Information and Computer Science, University of California, Irvine, Department of Information and Computer Science, University of California, Irvine, June 1990. TR 90-15.

[14] Y. V. Srinivas, "Category Theory Definitions and Examples," tech. rep., Department of Information and Computer Science, University of California, Irvine, Department of Information and Computer Science, University of California, Irvine, February 1990. TR 90-14.

[15] B. Liskov, "Data Abstraction and Hierarchy," in *(addendum to) Conference Proceedings, Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 1987.

[16] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Reading, Massachusetts: Addison-Wesley, 1979.